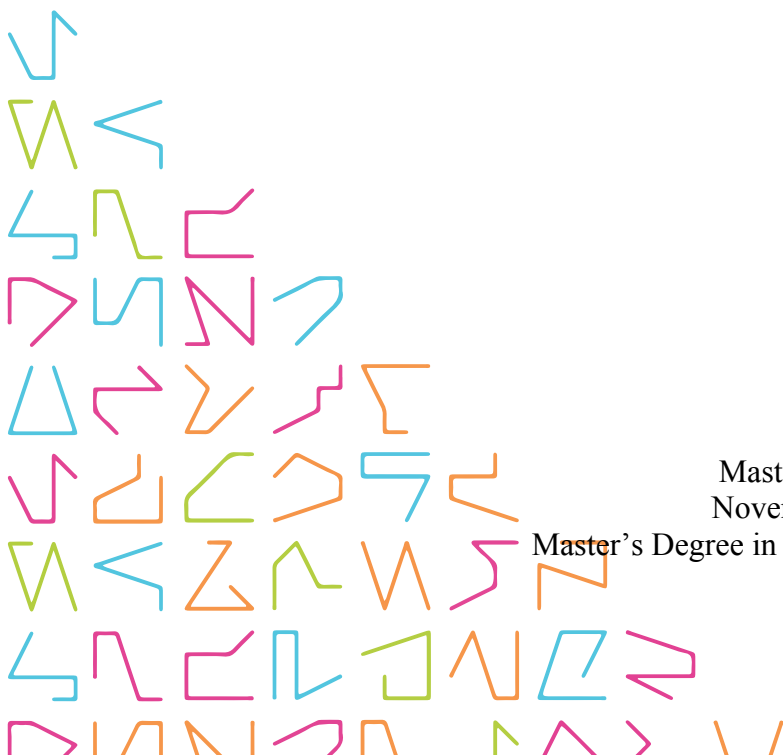


# **Improving Visibility of Test Results for Continuous Integration and Delivery Pipeline**

CASE: Liaison ALLOY Platform  
Development

Riku Halonen

Master's thesis  
November 2017  
Master's Degree in Information Technology



## ABSTRACT

Tampereen ammattikorkeakoulu  
Tampere University of Applied Sciences  
Master's Degree In Information Technology

Riku Halonen

Improving Visibility of Test Results for Continuous Integration and Delivery Pipeline  
CASE: Liaison ALLOY Platform Development

Master's thesis 43 pages, appendices 2 pages  
November 2017

---

In the modern software development cycle; testing, releasing and deploying software continuously has become an important factor of the process. Equally important is to be able to see the test reports from any given phase of the process, proving that the functionality and the over quality requirements of software product are met. Feedback in the form of test reports generated from automated tests part of the continuous integration and delivery pipeline help to react on possible defect immediately. Test reports are also an important part of the release documentation.

To be able track the quality of software continuously and the deviation between software builds, releases and deploy environments; results and other artifacts captured from automated tests has to be persisted in a centralized database or repository. This enables to create different views to data, such as reports and dashboards. This thesis addresses the need of capturing test results, possibly different formats, from continuous delivery pipeline to enable fast feedback for the project teams as well as reporting to different project stakeholders and to supplement release documentation.

The work is carried out for Liaison Technology Oy as part of the ALLOY integration and data management platform project. The goal is to improve the visibility of test result captured from continuous integration pipeline, so that it is easy to see status of projects' quality. The output and result from thesis is a design of a web service that provides upload and persisting of test results from deployment pipelines as well as queries over data the stored result.

---

Key words: continuous integration, continuous delivery, test automation, reporting

## CONTENTS

|       |  |    |
|-------|--|----|
| 1     | INTRODUCTION .....   | 6  |
| 2     | CONCEPTS AND BACKGROUND INFORMATION.....   | 9  |
| 2.1   | Continuous integration.....  | 9  |
| 2.2   | Continuous delivery .....  | 10 |
| 2.3   | Test progress monitoring and control .....                                       | 12 |
| 2.4   | Technology overview.....   | 13 |
| 2.4.1 | Jenkins.....   | 13 |
| 2.4.2 | Dropwizard.....  | 15 |
| 2.4.3 | ArangoDB .....   | 15 |
| 2.4.4 | Docker and containers.....   | 16 |
| 3     | TEST REPORTING SOLUTION FOR CONTINUOUS DELIVERY .....                            | 19 |
| 3.1   | Case Liaison ALLOY platform development.....                                     | 19 |
| 3.1.1 | Description of the case organization and project .....                           | 19 |
| 3.1.2 | Microservice architecture .....  | 20 |
| 3.1.3 | Description of continuous integration and delivery pipeline of the project ..... | 21 |
| 3.2   | Motivation for thesis .....  | 22 |
| 3.2.1 | Team and project level communication .....                                       | 23 |
| 3.2.2 | Compliance requirements.....   | 23 |
| 3.2.3 | Improving current reports .....  | 24 |
| 3.3   | Solution overview .....  | 24 |
| 3.3.1 | Use cases .....  | 26 |
| 3.3.2 | System under interest .....  | 28 |
| 4     | A DESIGN OF A BACK-END SERVICE .....   | 31 |
| 4.1   | Architectural overview.....  | 31 |
| 4.2   | Main system functions and features.....  | 33 |
| 4.2.1 | Storing test results .....   | 33 |
| 4.2.2 | Fetching test reports .....  | 34 |
| 4.3   | Data model .....   | 35 |
| 4.4   | API overview .....   | 36 |
| 4.4.1 | Upload API.....  | 37 |
| 4.4.2 | Projects API .....   | 37 |
| 4.4.3 | Test sessions API .....  | 38 |
| 4.5   | Security .....   | 38 |
| 4.6   | Deployment.....  | 39 |
| 5     | CONCLUSIONS .....  | 40 |

|  |    |
|--|----|
| REFERENCES.....                                    | 42 |
| APPENDICES.....                                    | 44 |
| Appendix 1. Example test report JSON document..... | 44 |

**ABBREVIATIONS AND TERMS**

|          |   |
|----------|---|
| Artifact | One of many kinds of by-products produced during the software development                         |
| B2B      | Business to business  |
| CD       | Continuous Delivery   |
| CI       | Continuous Integration  |
| CRUD     | Create, Read, Update, Delete  |
| DEV      | Development   |
| DSL      | Domain Specific Language  |
| Issue    | A software failure or fault report  |
| MFT      | Managed File Transfer   |
| Monolith | A server-side application with user interface, domain logic and database management capabilities. |
| PCI DSS  | Payment Card Industry Data Security Standard  |
| HIPAA    | Health Insurance Portability and Accountability Act   |
| QA       | Quality Assurance   |
| REST     | Representational State Transfer   |
| SDLC     | Software Development Life Cycle   |
| UAT      | User Acceptance Testing   |
| UI       | User Interface  |
| VCS      | Version Control System  |
| VM       | Virtual Machine   |

## 1 INTRODUCTION

Software companies and teams have broadly adopted a practice to build and test code changes frequently. This process is more commonly known as *Continuous Integration*, that has become a de-facto standard in software industry. The purpose of continuous integration is to verify code changes using an automated build and tests to detect problems and to capture defects as early as possible. This shortens the time to get feedback from the possible integration errors and therefore lowers the cost of defect because it is found in early phase of the Software Development Life Cycle (Hambling & Morgan & Samaroo & Thompson & Williams 2015).

*Continuous Delivery* introduces one extra step on top of continuous integration process allowing software to be deployed at any time into any environment. One of the key principles in continuous integration and delivery is that when somebody makes a change to the software, anybody can get a fast feedback about functionality and health of the system to determine its readiness for production. Continuous delivery can be achieved by deploying into *production-like* environments frequently and running automated tests to ensure software will work in production (Fowler 2013). The quality of software and production readiness can be determined by capturing test results deployment pipelines and generating reports, metrics or dashboards to support decision-making.

To be able track the quality of software continuously and for instance deviation between builds or releases; results and other artifacts captured from automated tests needs to be persisted in a centralized database or repository. This enables to create different views to data, such as reports and dashboards. Continuous delivery brings another aspect to this, which is to track deviation of results between environments. Figure 1 illustrates an example of such pipeline, where a continuous integration server is responsible of building and running automated tests as well as deploying artifacts into different environments as software moves forward in the pipeline.

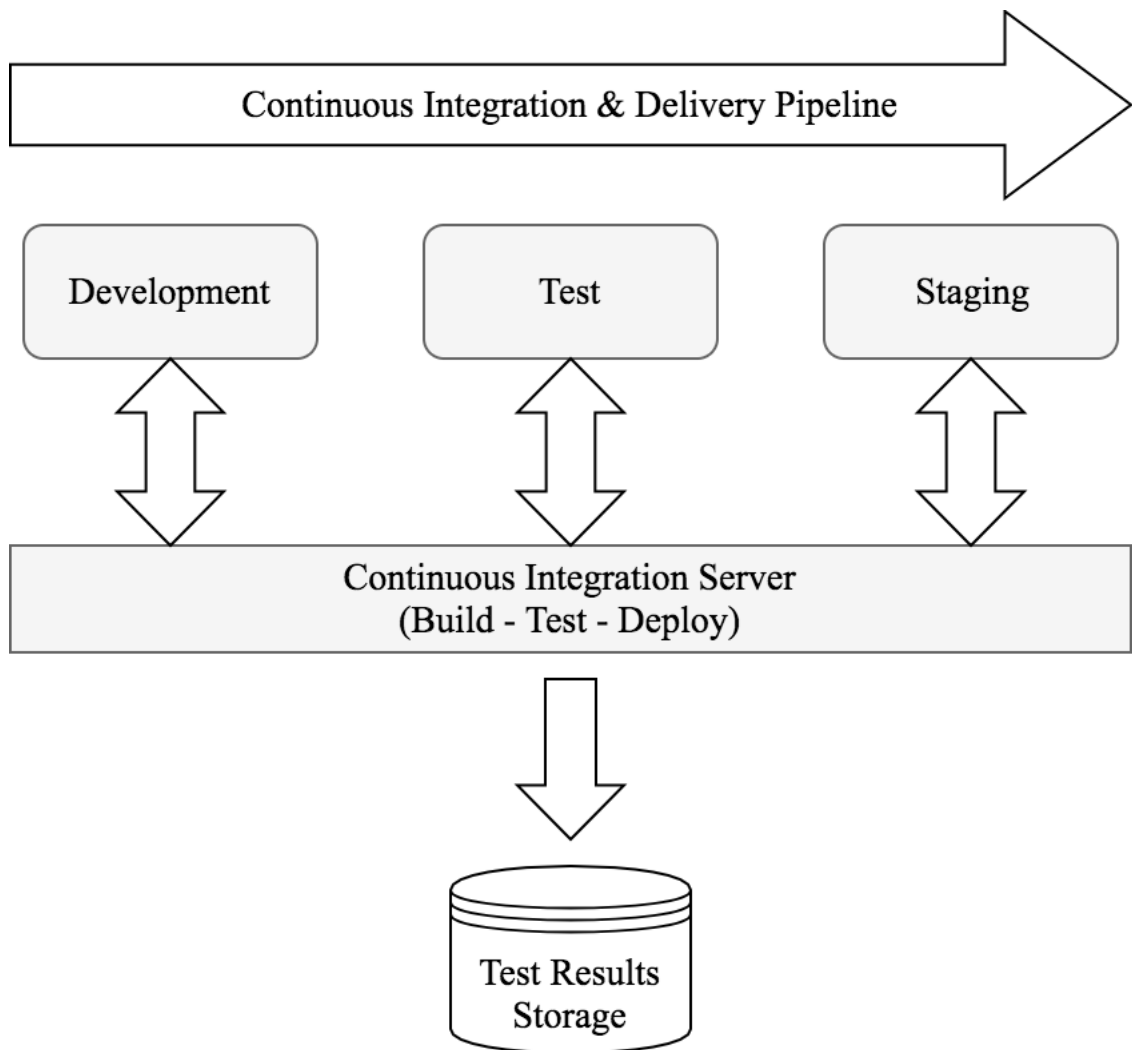


FIGURE 1. Continuous delivery overview

Interests towards test reports and metrics may vary between project stakeholders. Developers might only want to see that the unit test results of their components from latest test run and therefore may not have many requirements for reporting. QA departments on the other hand are usually interested in the overall quality concerning all related software components of a project. QA is usually looking for history data for instance to be able to compare results between software builds or releases. Project management might only be interested in quality metrics.

Organizations may have other liabilities as well due to why test reports are needed and persisted for later use. For instance, a company and its manufactured software products might have been certified by some standards to be able to handle confidential information such as health care or credit card data. To be able show compliance with the standards there usually needs to be proof that testing has happened, that is, a test report.

This thesis addresses the need of capturing test results, possibly different formats, from continuous delivery pipeline to enable fast feedback for the development teams as well as reporting to different project stakeholders. This can be particularly challenging in a software platforms that consist of multiple micro-services, varying technologies and worked out by global team, but also important to formulate outline of platform quality. The practical part of the work is carried out in a development department of a mid-sized software company, Liaison Technologies Oy. Thesis has the following goals:

- Study what are the requirements for test reports and metrics for continuous integration and delivery in the case organization.
- Design a system that implements uploading and persisting of test results from deployment pipelines as well as queries over data.
- Enable building of user interfaces by providing necessary APIs from back-end service.

Ultimately, the goal is to improve the visibility of test result captured from continuous integration pipeline, so that it is easy to see status of projects' quality at a glance and to provide fast feedback to teams.

The scope of this thesis is mainly focus on defining the database and back-end service design as well as implementation of later mentioned. The final system shall also include a front-end application for the end users, but it is not in scope of this thesis. It is still expected that the outcome of this thesis will be later used to build web application on top of the Restful API.

The document is organized as follows. Chapter 2 addresses concepts and theoretical background of this thesis. Chapter 3 focus on introducing a case project and the practical work. Chapter 4 walks through the design and implementation that was conduct to a test reporting system for continuous delivery. Chapter **Error! Reference source not found.** focuses on analyzing the implemented solution and draws final conclusions.



## 2 CONCEPTS AND BACKGROUND INFORMATION

As this thesis builds on top of continuous integration, continuous delivery and test reporting, the purpose of this chapter is to give an overview of these three topics of software engineering. This chapter will also introduce some of the technologies used in the sections discussing about the practical development work conducted as part of this thesis.

### 2.1 Continuous integration

For many years, a typical characteristic in a software projects was that during development process, application might have had been in a non-functioning state and teams were working a significant time on an unusable code. There might have been frequent commits of code and some level of unit testing happening, but no one was trying to run the application until later phase of the project (Humble & Farley 2011, 55). This might had happened because of various reasons, but most typical being that software integration and acceptance testing were scheduled to happen only after the development phase. This resulted into situations where integration and testing phase took unpredictably long and in worst case it was seen nearly at the end of the project that application did not meet its requirements (Humble et al. 2011, 55). Continuous integration, which has now been widely adopted by the companies, has removed these issues and making sure that software being developed is in working state all the time.

Figure 2 illustrates a typical continuous integration process. In continuous integration, every time when somebody commits a change into version control system, application is immediately built and automated tests run against it. Teams get feedback in the form build status and test report. If there were build or any test failures, teams should fix the problems immediately, before committing any new code. Continuous integration therefore has indisputable benefits. Assuming a comprehensive automated test suite has been written for the application and runs for every commit, with continuous integration, software is proven to work always and teams will immediately see when it breaks and can react upon on it (Humble et al. 2011, 56).

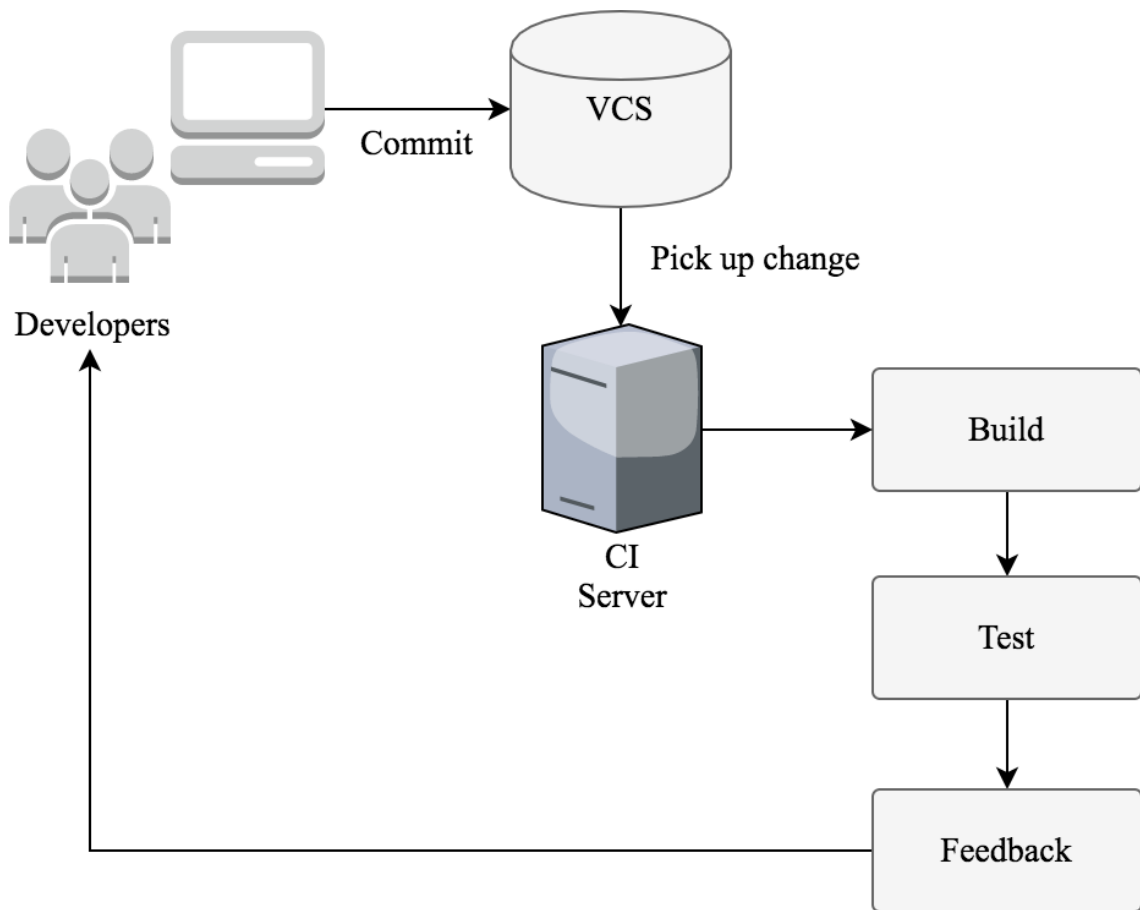


FIGURE 2. A simplified continuous integration flow

The most essential parts in continuous integration are visibility and communication in a way that teams can see for example the state of the builds or latest test results. For maximum visibility, these should be provided in the form of web pages and aggregated dashboards. These are especially useful if teams are not co-located when communication can be more difficult compared to a project teams sitting in a space office space (Fowler, 2006).

## 2.2 Continuous delivery

If continuous integration was about ensuring that software remains in working state throughout the project lifecycle by building and testing software from every change, continuous delivery is about enabling releases and deployment of software to any environment at any time. According to Fowler (2013) continuous delivery software development discipline has the following characteristics:

- Teams working on software are committed to keep software deployable throughout its lifecycle and prioritize fixing problems over new features.
- Anybody can get fast and automated feedback on the production readiness of their software.
- It is possible to perform an on-demand deployment of any version to any environment.

In other words, continuous delivery can be achieved, in addition to following continuous integration development practices, also continuously deploying executable into production-like environments. Benefit of this is that, teams can detect possible production issues early, which then leads to reduced deployment risk (Fowler 2013).

Continuous delivery process is usually modeled as *Deployment Pipeline* that is implemented into CI server, for instance. A deployment pipeline process involves *stages* from building the software followed by multiple test runs and deployments into different environments (Humble et al. 2011, 109). Figure 3 shows a generic deployment pipeline capturing the basic idea. A real pipeline would have more or different steps specific to each project's process of delivering software.

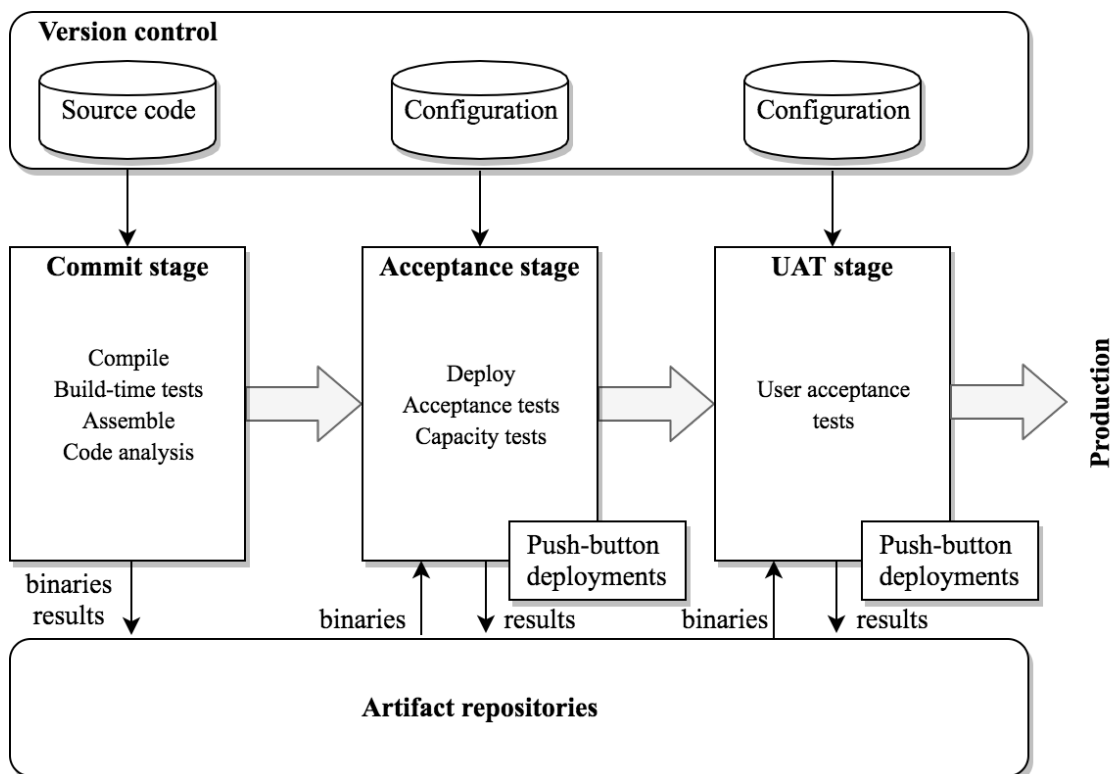


FIGURE 3: Deployment pipeline (Humble 2011, 111)

Process starts from left of the pipeline developers committing changes to version control. As already discussed in chapter 2.1, at this point continuous integration server triggers the first stage of the pipeline that compiles the code, runs tests and analysis code statically. If these steps pass, build and test artifacts are pushed to repository otherwise stage fails, feedback is sent to team and responsible person should take an action to fix issue. In any case test reports should be available to be able to analyze test failures. (Humble et al. 2011, 111)

Second stage of pipeline, referred as acceptance stage here, is usually triggered automatically and only if first stage passes. This typically includes longer-running functional and non-functional tests usually implemented by QA department. At this point software is usually deployed on one or more environments. To speed up test execution and to be able to get feedback faster, tests are usually split into several jobs that run parallel (Humble et al. 2011, 111).

After acceptance gate, deployments are usually not performed automatically; instead different teams do self-service deployments into environments owned by them. In Figure 3 such environment is referred as UAT, which could be used to do some final verification of customer use cases. (Humble et al. 2011, 112) It is important to notice here that teams must be constantly able to see the status of builds and tests, so that they can be confident enough to deploy a “healthy” version to their environment.

### **2.3 Test progress monitoring and control**

Software testing is systematic exploration of a system or a component to find and report defects. There are several types of reports testing can produce, which main purpose is to communicate the test results and findings to the project stakeholders to support decision making for making the software release or, if further testing is required and release must be delayed (Hambling et al. 2015). Probably the most common report artifact is a test execution report showing passed and failed test cases.

The purpose of monitoring test progress is to provide feedback and visibility of test activities. The data can be collected for instance from database already holding the execu-

tion results and formatted into different reports and graphs (Hambling et al. 2015). Common test metrics, according to Hambling et al. (2015) include:

- Test case execution metrics including number of test cases run/ not run, and test cases passed/ failed.
- Defect information including defects found and fixed or failure rate.
- Test coverage metrics (e.g. branch, statement or feature coverage).

Metrics are often visualized in graphical form. A trend has been to move towards dashboards, which display relevant metrics on a single page or screen.

Test control is one of the most important activities that utilizes data collect by test monitoring and uses that information to drive testing efforts. Test control is needed to make sure exit criteria for testing is met which can be e.g. 100% pass rate of tests or 100% feature coverage (Hambling et al. 2015).

## 2.4 Technology overview

### 2.4.1 Jenkins

Jenkins is an open-source automation server, which can be used to automate all sorts of IT tasks, but is mostly used to automate building, testing and deployments of software. With its cross-platform support, extendibility capabilities and active community support, many companies have chosen it to build and test their software projects continuously. Jenkins server also allows to continuously delivering software by providing a capability to define build pipelines and integration with number of test and deployment technologies (Jenkins 2016).

Jenkins Pipeline is a recently added feature into Jenkins enabling continuous delivery. In practice, Jenkins pipeline is suite of plugins and an extensible DSL defining pipeline as code. Figure 4 illustrates an example model of a pipeline along with its main building blocks. A *Stage* is conceptual subset of a pipeline, for example: “Build” or “Test”. A *Step* is a single task within a stage defining steps *what* Jenkins should do. This can be for example a shell command.

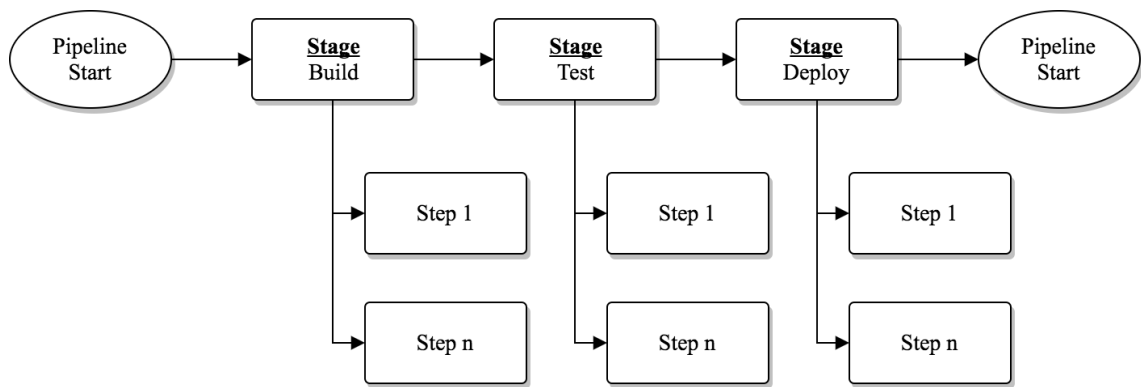


FIGURE 4. Jenkins deployment pipeline (Jenkins Pipeline)

Figure 5 shows an example Pipeline DSL. As delivery pipeline can be defined as code, it makes possible to store it into VCS along with the projects' source code and versioned.

```

pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        sh './gradlew build'
      }
    }
    stage('Test') {
      steps {
        sh './gradlew test'
      }
    }
    stage('Deploy') {
      steps {
        sh './gradlew publish'
      }
    }
  }
}

```

FIGURE 5. Jenkins pipeline DSL

Jenkins was selected to be part of this thesis as it was already used in the company to implement continuous delivery. Therefore, it was natural choice to start integrating also test reporting into Jenkins and pipeline DSL.

### 2.4.2 Dropwizard

Dropwizard is an open source Java framework for building RESTful web-services. In fact Dropwizard is a collection of widely used Java libraries such as Jetty, Jersey or Jackson and solves the problem of manually adding and configuring different libraries when building a web service (Dallas 2014).

Dropwizard uses Jetty HTTP library to embed a HTTP server into project. Java projects utilizing Dropwizard do not require an application server such as Tomcat or JBOSS, but instead they have a *main* method, which start the HTTP server. This makes for instance deployments and maintenance much easier. For building RESTful web applications, Dropwizard bundles Jersey. It allows writing Java class that map HTTP requests to Java objects. Dropwizard has good support for JSON as a data format. For serializing JSON into Java objects and vice versa, it uses Jackson library. (Dropwizard 2017)

In this thesis Dropwizard framework is used to implement a RESTful web application that serves APIs to import test results files into database and to query data for reporting purposes. There is much more as well that Dropwizard framework includes, for instance JDBC to access relational databases from Java, but this is not in scope of this thesis. Instead a NoSQL database is used to store data.

### 2.4.3 ArangoDB

ArangoDB is an open source NoSQL database that allows storing key/ value, document and graph data and queried with SQL-like language. Database uses JSON as a storage format, but internally it uses its own compact binary format to store and serialize data (arangodb.com 2017). JSON documents in ArangoDB are stored and grouped into collections. In comparison to traditional relational database, collections can be compared to tables and documents to rows.

Whereas in relational database system, where columns are needed to be defined before records can be stored into table, also known as schema, ArangoDB doesn't have this requirement. It is said to be schema-less meaning that there is no need to define what attributes documents should have. Every document in a collection may have completely different attributes (arangodb.com 2017). Like relational database where table records can be altered, also in ArangoDB, documents can be updated or deleted and attributes modified.

The data stored inside ArangoDB can be accessed, queried and altered multiple ways, using the ArangoDB Query Language (AQL), using the HTTP REST interface or using a programming language specific driver. This thesis focuses on the ArangoDB Java driver to communicate with the database.

#### **2.4.4 Docker and containers**

Docker is an open source technology designed to ship, deploy, run and scale application easier and especially targeted to cloud-based solutions. Docker allows to package applications with all its dependencies into a distributable image that can be deployed to any Linux or Windows machine and thus being sure that application behaves similarly always (opensource.com 2017). This creates several advantages by speeding up development cycle and shortening time-to-market. When distributing applications as Docker container images, the same image can be used in development, for testing and also in production. Docker is very similar to virtual machine. The main difference is that Docker container images do not contain the full virtual operating system, but allows applications running inside a Docker container to use the same Linux kernel as the host system (figure 6).



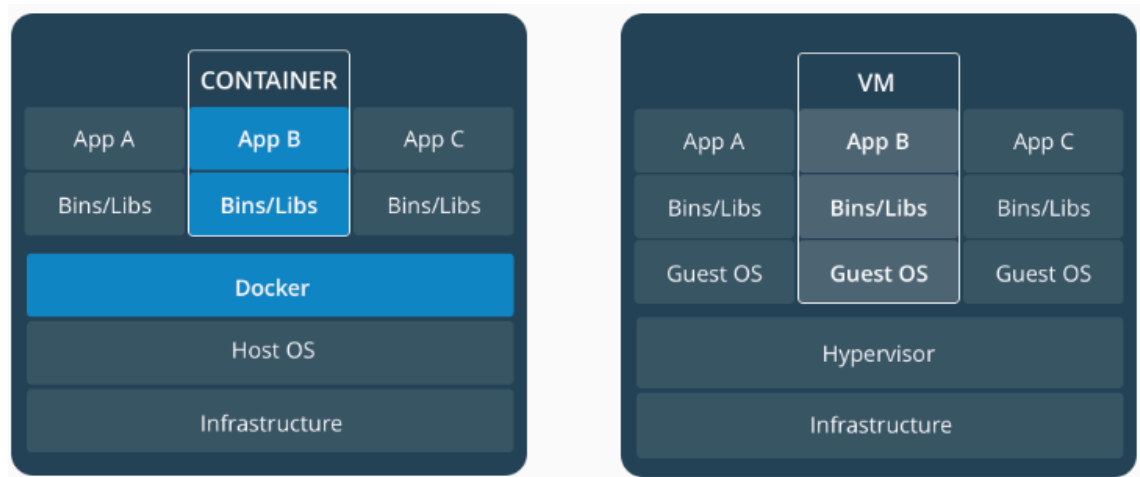


FIGURE 6. Containers and virtual machines compared (Docker 2017a)

A Docker image is a package that mainly includes the application binary to be executed, runtime, system tools and libraries and settings. A container is a runtime instance of an image when it is executed and loaded into memory. A container requires always a host system, usually a VM where it is executed but it runs completely in an isolated environment with possibility to access host resources such as file system or ports (Docker 2017b).

Docker installation comes with a complete toolchain for building, running and orchestrating containers. The container environment and what is put inside the container is defined in a *Dockerfile* as shown in the following.

```
# Use official OpenJDK as a parent image
FROM openjdk:8-jre-alpine

# Copy the application into image's /app directory
COPY build/*.jar /app

# Expose port 80 to the outside world from the container
EXPOSE 80

# Command to run when container is executed
ENTRYPOINT ["java", "-jar", "application.jar"]
```

Once the Dockerfile has been written to instruct Docker how to package the application, the image can be built using *docker build* command and executed using *docker run*. Container is then loaded into memory and the example application can be reached from port 80 in host IP address or domain name.

### 3 TEST REPORTING SOLUTION FOR CONTINUOUS DELIVERY

#### 3.1 Case Liaison ALLOY platform development

This thesis was conducted for an organization involved in a development work of a cloud-based integration and data management platform. The case project and its continuous integration and delivery processes are introduced in the following chapter as well as motivation for this thesis.

##### 3.1.1 Description of the case organization and project

ALLOY is a cloud-based integrations and data management platform by Liaison Technologies. ALLOY platform covers such integration patterns as B2B or MFT and use cases that may occur between business partners for instance customers, suppliers, and banks or logistics providers.

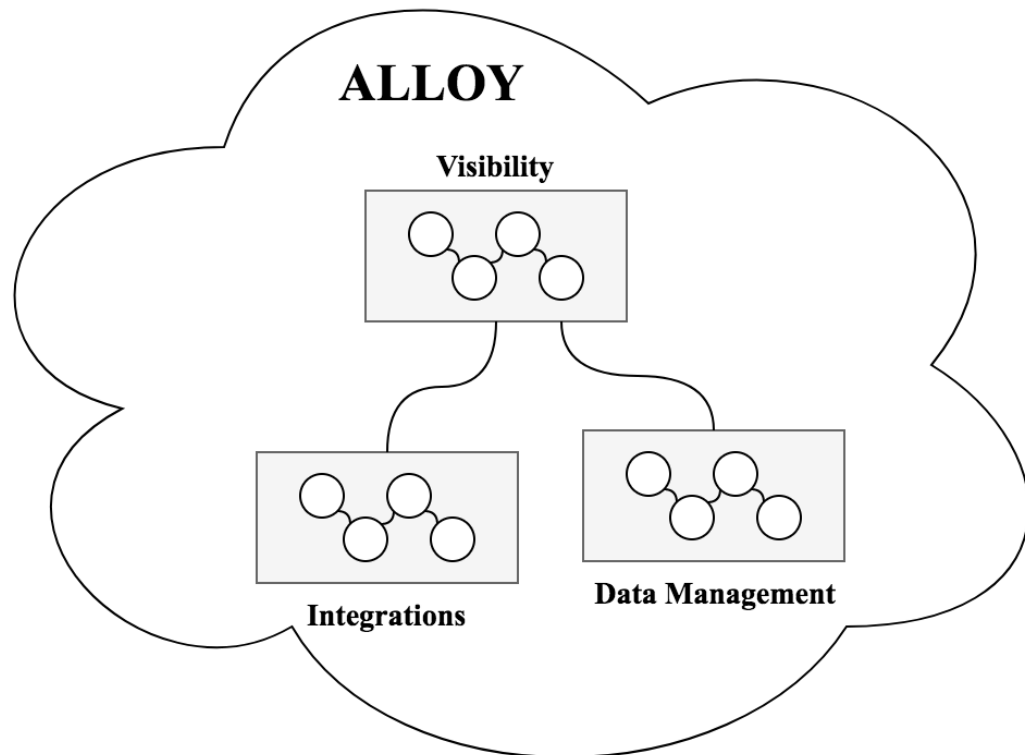


FIGURE 7: Liaison ALLOY Platform

A typical use case is a purchase order happening between two companies, which systems are incompatible with each other, but still would need to be able to communicate to each other. In this situation ALLOY cloud sits in between of these two trading partners receives a purchase order from source application and routes it seamlessly to target application. Optionally there might be some conversions done inside the cloud, for instance from file type to another (liaison.com 2017). Another feature of ALLOY integrations is Managed File Transfer (MFT). MFT refers to a service managing secure transfers of data over a network, providing support for multiple secure transfer protocols such as SFTP or AS2.

The other two layers of ALLOY platform as Data management and visibility of ALLOY employ big data technology for storing and computing data. Customers can connect to data repositories using APIs and alter or analyze data as needed. Visibility layer provides different user interfaces or dashboards for example to monitor data flows or integration status as well as different reports (liaison.com 2017).

### **3.1.2 Microservice architecture**

The ALLOY platform is built on the microservice architecture. Fowler & Lewis (2014) define microservice architecture as a model of developing a single application as suite of small services. Each application runs then in its own process communicating to each other, often over http API. Services and applications may build on different technologies, written in different programming language and use different persistence layer technologies.

A common characteristic of microservice architecture is that large applications are split into UI, server-side logic and database layer both on technology and team level. This creates advantages compared to monolithic architecture where applications are built as a single unit. A typical monolith enterprise application consists of three parts: user interface, database and a server-side application to execute business logic communicate with database and populate views. Monolithic applications can be scaled, build, tested and deployed automatically, but making a small change to such application requires rebuilding the whole application as all functionality is put into a single process. In a micro-

service architecture each element of functionality is put into a separate service. (Fowler et al. 2014)

Continuous Delivery has a key role in microservice-based architecture, because separating functionality into smaller individual services makes it possible to release more often. A prerequisite for this is that testing has also been automated that the test coverage is on a decent level so increase confidence that software is really working. Promotion of tested software up the pipeline requires deployment automation to each environment (Fowler et al. 2014).

### 3.1.3 Description of continuous integration and delivery pipeline of the project

The continuous integration and delivery pipeline process in the case organization and project is illustrated in figure 8. As part of the process there are several environments compiled code is being deployed for testing various things during the process. For simplicity, local development environments as well as production environment are left out from the picture.

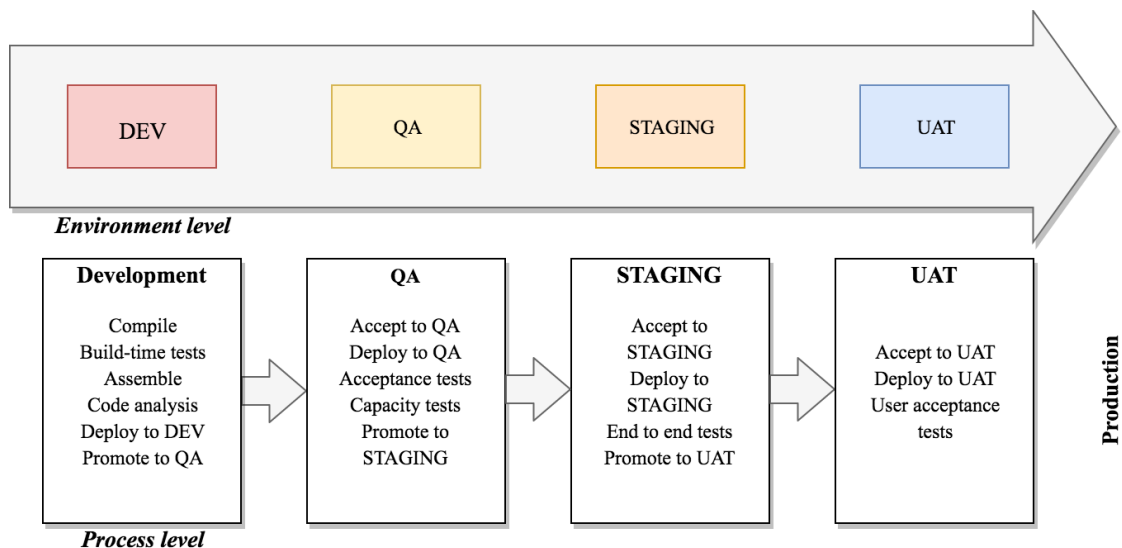


FIGURE 8. Continuous integration and delivery process pipeline

Starting from the left, there is a DEV environment that serves the need of deploying latest changes. This usually happens multiple times a day as the developers work on the features and fixes. Before deploying to DEV, CI server always executes development

time tests. After code has found to be working in DEV, it can be promoted and accepted to next environment (QA), where acceptance testing including all platform features and components. This is because; development time testing commonly focuses on a single project or component. Tests executed in QA environment are typically provided by QA department that treads platform as one and assesses all it capabilities and features. At this point QA department can either sign-off the deployed version and promote it or send it back to development. STAGING and UAT environment are meant for even more specific testing, before production deployment.

As shown in figure 8, between each environment deployments there are *promote* and *acceptance* stages where manual user input is mandatory. This is a key concept in concept in continuous delivery that pipeline allows frequent deployments, but can be stopped anytime (Fowler 2013).

### 3.2 Motivation for thesis

Getting an immediate feedback from continuous integration and delivery pipeline to assist decision-making whether deployment should be promoted or accepted to next environment, a feedback is necessary, in a form of reports, showing the results of testing. Figure 9 shows the key drivers identified of this thesis and why there was a need to improve the current reporting and visibility of results.

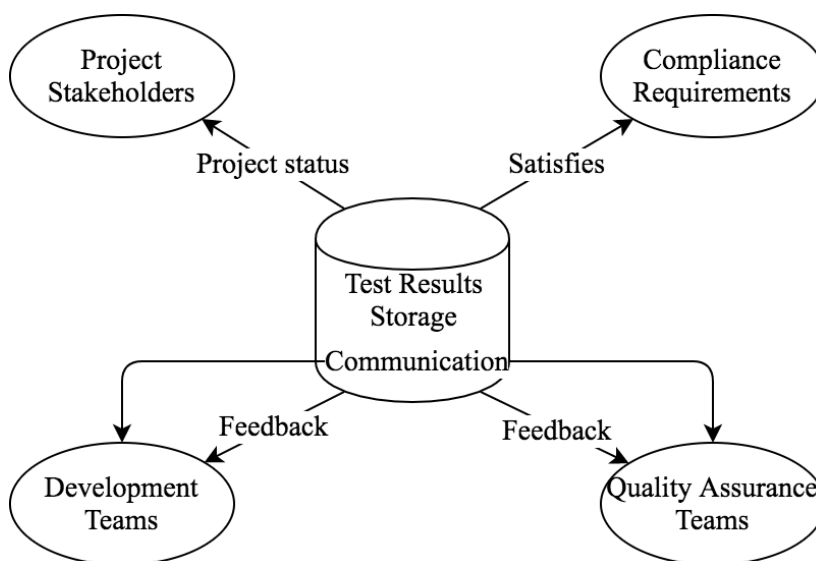


FIGURE 9. Test reporting key drivers in the case project

### 3.2.1 Team and project level communication

In a software development project that is following continuous integration and delivery discipline, development and quality assurance teams should work closely together. Test reports provide an important tool for communicating project status between DEV and QA teams as well as providing visibility to other stakeholders. For example, without any test reports it would be difficult for a QA person to see, whether code has been unit tested by a developer, if there is no evidence about testing. In the case project, where QA department does the sign-off for projects based on test results, such visibility to developer tests is necessary.

### 3.2.2 Compliance requirements

As Liaison ALLOY platform deals with a sensitive data, it has been certified by compliance standards such as PCI DSS or HIPAA. The PCI DSS defines standards to support payment card data security and guidelines for implementing process to prevent and detect security incidents (PCI Security, 2017). HIPAA standard protects sensitive patient data, which means that any company dealing with protected health information must comply with the rules set by the standard. That said, HIPAA standard concerns both *covered entities* providing treatment, payment and healthcare operation as well as *business associates* with access to patient information or providing support in operations (HIPAA, 2017).

Companies which products have been certified to be PCI DSS or HIPAA compliant such as Liaison ALLOY platform are being audited regularly to make sure the compliance requirements are met. Standards set requirements to SDLC processes that reflect to software quality assurance. This sets requirements for test reporting as well because it must be able to prove and have evidence of:

1. What software and versions have been tested?
2. How the software has been tested?

### 3.2.3 Improving current reports

At the time when work for this thesis was started, reports produced from automated testing where mainly persisted to satisfy compliance requirements and stored as *static* HTML pages. For example tracing back the root cause of a failed test case was quite difficult as there wasn't any linking between build and issue management system or source code repository. Due to the fact that there wasn't any database where test results would have been store and because those were saved into file system as HTML pages, targeting any search operations to data or finding any results history was impossible. These were the primary needs why this thesis work was initiated.

## 3.3 Solution overview

Many of the products available either open source, proprietary or commercial providing solutions for test reporting, are often test management tools that offer more than just the reporting functionality. A test management tool is a software that can be used to guide the whole testing process from how testing is to be done through planning and execution up to the reports. Test management tools usually share a common feature set and challenges, which include:

- Ability to track feature test coverage which requires that requirements of the software under test are written into same tool or imported from other system so that they can be linked to test cases. This requires a commitment from the whole organization and especially from requirements management that the software requirements are well defined in the tool and kept up to date.
- To be able to use them for reporting, test cases must be planned and documented into test management tool. As for the requirements to be able to get the most out of the tool all parts of the development organization must be committed to use the same tool for test planning. This may be obvious for a QA person, but software developers usually object any extra documentation, if they don't see the benefit.
- Test execution and planning process built into test management tools is usually driven by the model where features are first developed and then handed over to QA teams for testing. In continuous delivery process, testing responsibilities are



spans the development and QA teams. In this process a full-featured test management tool can start to be a burden, as it cannot fit into fast and continuous testing cycles. In continuous delivery, it is more important to bring visibility to testing for improved communication and fast feedback.

The starting point for this thesis was to design a test reporting solution that supports continuous testing process and is targeted mainly for automated testing. A traditional test management tools did not seem to provide a lightweight solution and due to the above-mentioned challenges adopting such tool would have needed support and commitment from the whole organization.

Target was set to design and partially implement a solution that would improve the way that test results and artifacts were persisted in Liaison ALLOY platform development, so that projects' continuous integration and delivery pipelines would produce better feedback reports. Figure 10 shows the sub-systems that were part of the reporting solutions design work.

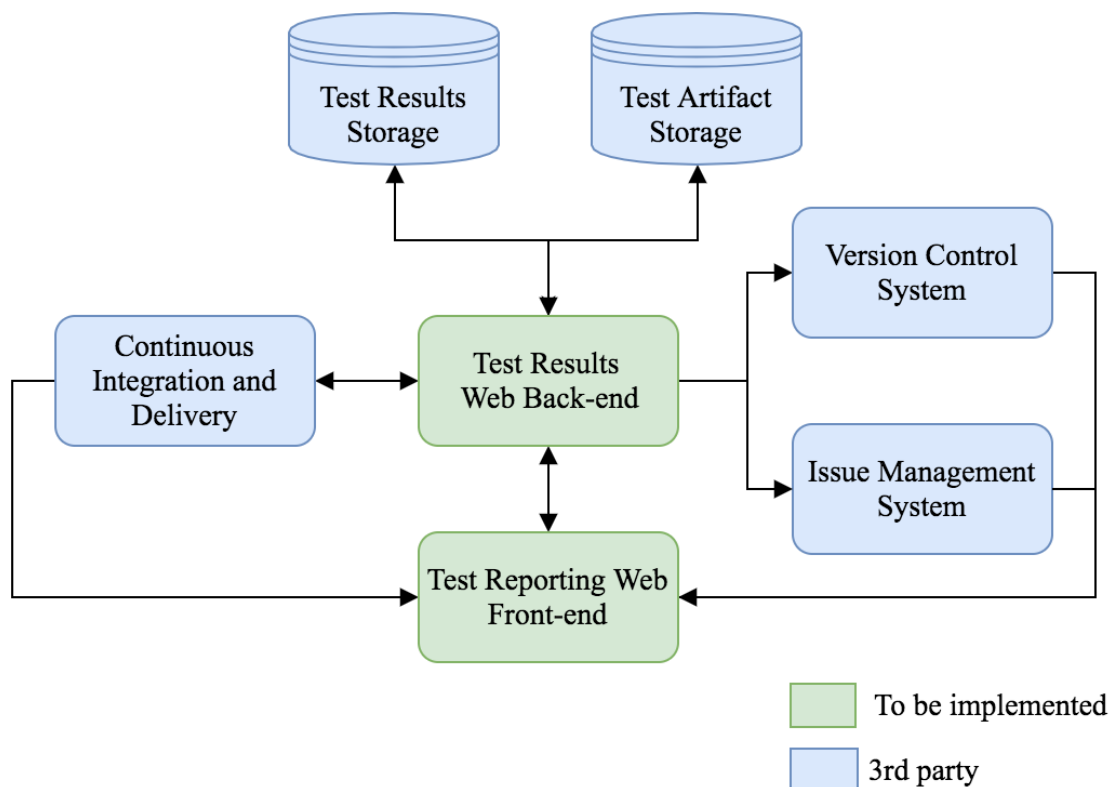


FIGURE 10. System overview

As one of the requirements for the improved reporting solution was to be able to search test results using various query filters or browse history, it was obvious that data must be persisted into database. Test results and artifact storages in figure 10 refer to persistence layer where test results and attachment captured from testing are stored.

Already in the beginning of thesis work, it was identified that there was a need to have a back-end web service that would act as an interface between applications and data storage layer. One of the main reasons for this selection was that it would allow to build any number of different reporting user interfaces and dashboards to visualize test results data. In addition to reusability, separating API from the client-side applications was good for modularity as applications could be built and maintained separately.

To be able to trace back later a failed test case to a particular code change being a possible cause for the failure, linking between test results and version control system was required. Information about the change itself such as project's code branch where software under test was built or commit id was possible to capture from build system. Another important requirement to have linking between a failed test case result and an issue report was to be able to have visibility when issue gets resolved and related test case starts to pass.

As the primary driver for the solution was to have extensive test reports to be generated from the continuous delivery pipeline, it would user be the primary "user" for the system, as the results would be imported from the Jenkins pipeline through the back-end API. For the *human* end-user interface plan was to build client-side application or applications for visualizing test reports aggregating stored data.

### 3.3.1 Use cases

The use case diagram in figure 11 illustrates system use cases this thesis would focus on. There are two types of actors that interact with the system: A *system* user, the Jenkins continuous integration server which would only upload test results from its build and deployment pipelines and the *human* end users that would view or later alter results manual though a web user interface.

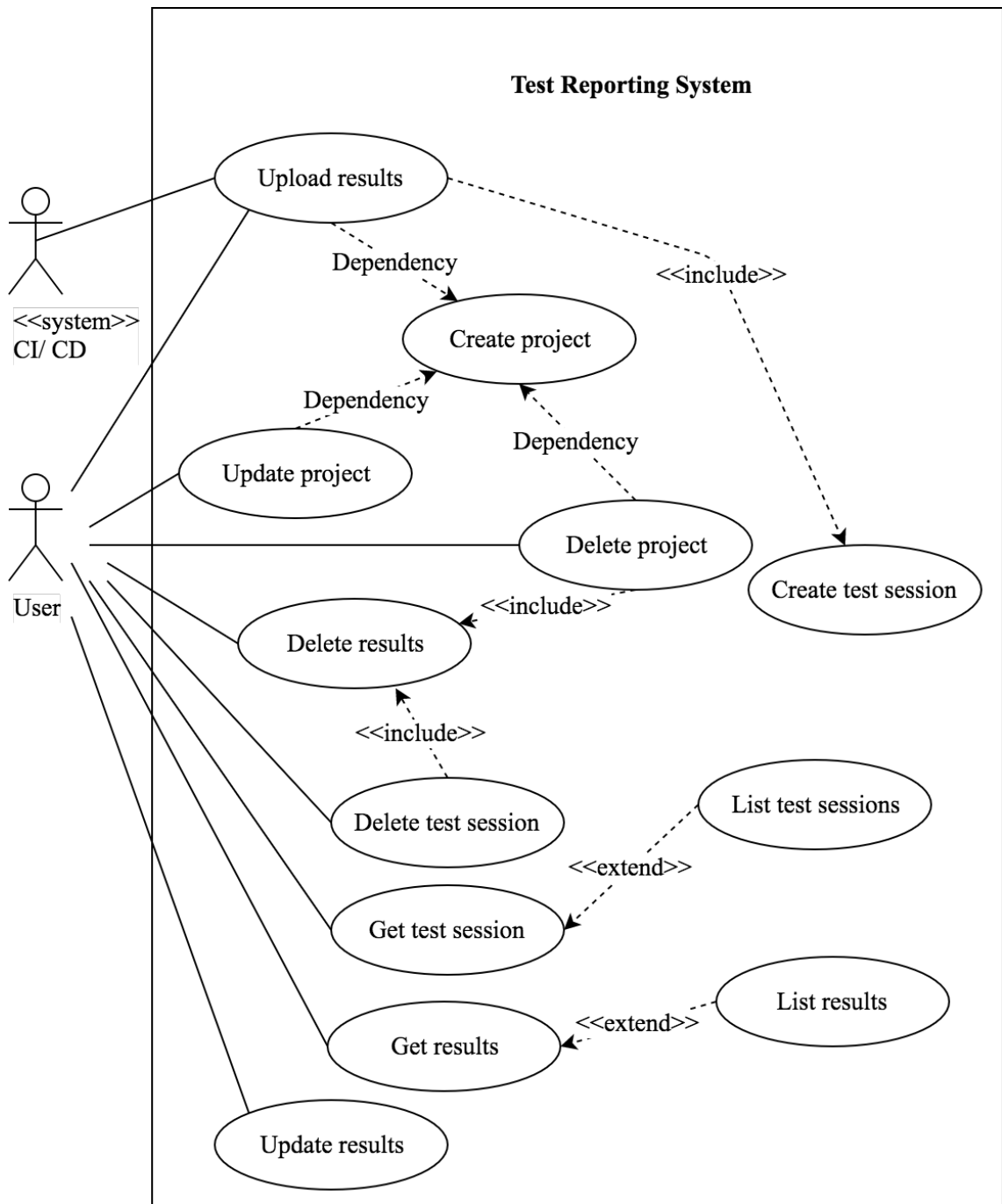


FIGURE 11. System use cases

Figure 11 refers to terms *project*, *test session* and *test result*. As these terms are used later in this document multiple times, definition of these terms in the context of this thesis is clarified in the next bulleted list.

- **Project** - A software project having its source code stored into code repository and to which uploaded test result files link to. A project might have been target of testing or it just relates to it other ways.

- **Test session** - A test session represents individual test run and context in which testing was performed. Test session may have properties such as title, test type and other build context information.
- **Test result** - Test result represents a result document uploaded into system.

Because of the microservices architecture used in ALLOY platform, where services and applications part of the final platform product can each have an individual lifecycle, there was a need to have linking also between projects and test results in the test reporting system. This is also a common design in all well-known continuous integration server products where a build, test and deployment pipeline is usually tight to a one software application or service project. This generated a requirement for the system that user must create a *project* entity, before uploaded results can be linked to it. Other management use cases identified related to projects was delete and update operation, where delete would also remove linking and optionally results itself.

Even though the primary actor for uploading test results into system would be the continuous integration server, there might be cases when users need to upload results manually for any particular reason. One reason might be to import new results into already existing test session for instance when automated upload has been failed. This action would occur by interacting with the system via front-end web UI.

After test results have been imported to system data must be able to query from UI that would later be implemented to generate the actual test reports. Query operations were categorized into a *get* operation that would return a single test session or result entity and a *list* operation that would return a group of entities.

### 3.3.2 System under interest

The implementation scope of this thesis was narrowed down to focus on the back-end service that will be allows uploading test results from continuous delivery pipeline, persists them and acts as an API for front-end applications to be implemented later. The sub-systems that were mainly scoped to be part of the thesis work are shown in figure 12.

Test frameworks used for unit, integration or any other type of automated testing, always generate a result file as an output that includes for example pass or failure status of executed tests. The format of a result file may vary between test frameworks, but the most common format that all well-known frameworks support is xUnit XML format. For example all Java or JavaScript test frameworks that were used in the case platform project did output xUnit formatted result file. Thus, the goal was to capture these result files that were captured from Jenkins pipelines' test stages and persist them into data storage for later use.

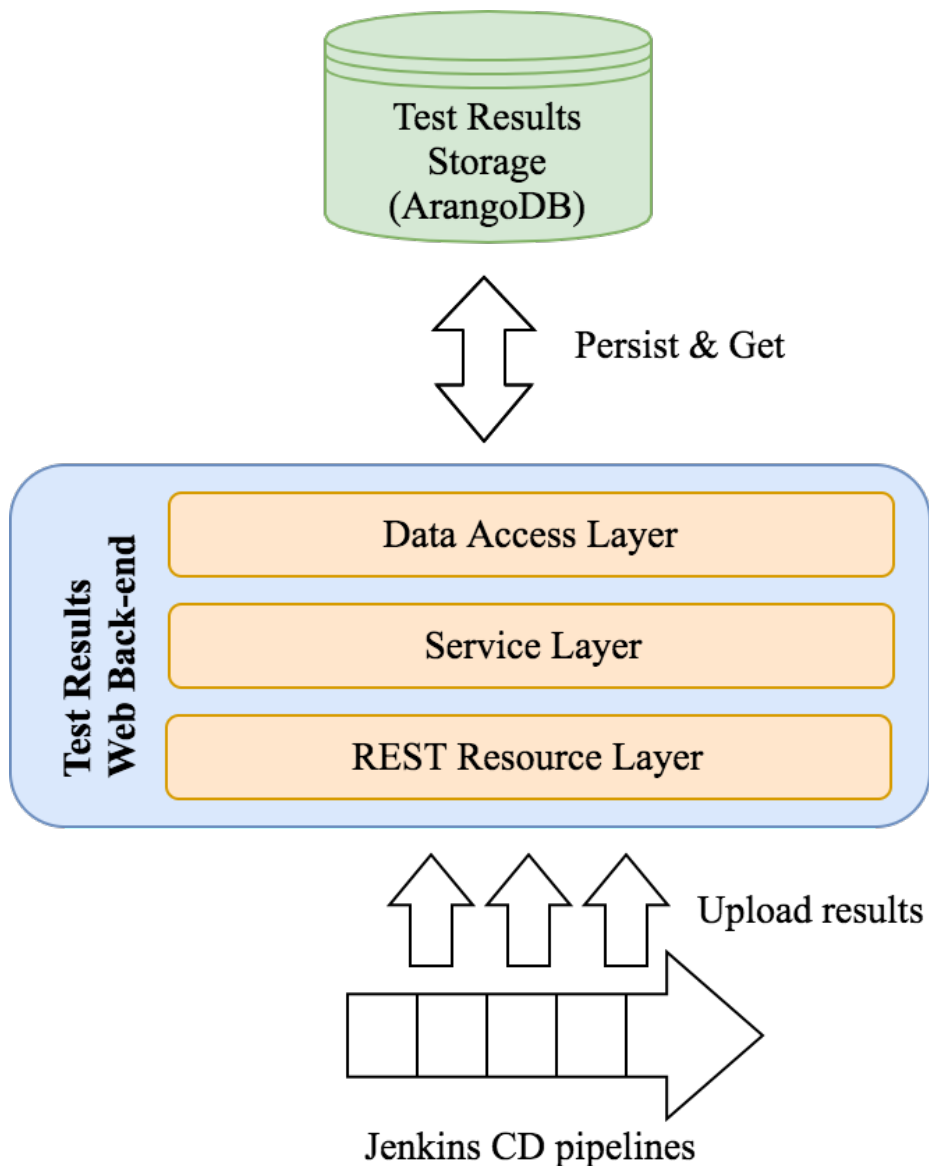


FIGURE 12. System under interest

The final system that would also later include the user interface was chose to follow the 3-tier architecture pattern. In a web application environment, a 3-tier architecture consists of application tier, web tier and data store tier representing the physical deployment of sub-systems. Figure 13 shows UI and Jenkins CI server on the application tier. These clients act as consumers to RESTful web service sending HTTP requests to web tier. After receiving a request from client, it is handed over to business logic layer. This layer includes functionality that the application must do for the domain, such as processing data and input validation (Fowler 2002). In this thesis context, layer is responsible of processing input result files and transforming them into correct format.

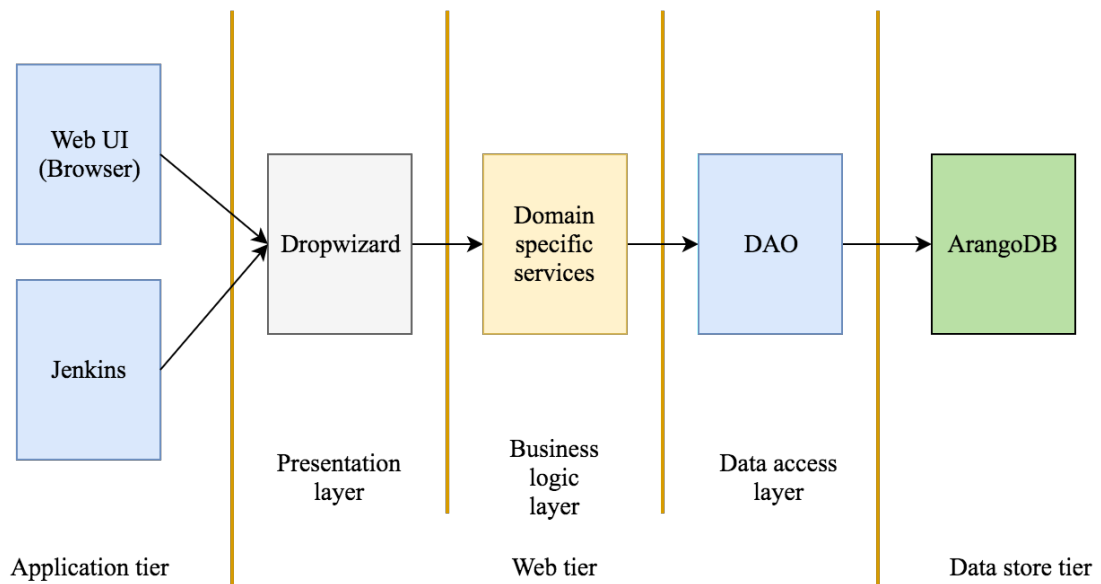


FIGURE 13. 3-tier architecture model and layers

The data access layer provides the interface to the data store tier and handles communication with data store systems on behalf of the web tier application. ArangoDB was selected as data storage for the test results. This was it was already being used in the organization and its capabilities to create relationships between documents and query them efficiently using graphs. Other reasons for selecting a NoSQL database was that test reports are documents per se and the fact that it would provide a bit more flexibility as data is not required to be normalized too strictly beforehand. As ArangoDB handles and stores data as JSON documents, test results data was required to be mapped from input format into JSON documents.

## 4 A DESIGN OF A BACK-END SERVICE

### 4.1 Architectural overview

Layered architecture diagram of the back-end test result service is presented in figure 14. It uses Java programming language based Dropwizard framework to build a RESTful application. As Dropwizard embeds HTTP server (Jetty), web application is packaged into an executable JAR and started in its own servlet container. This makes the deployment of web application easier and perfect candidate to be packaged inside a Docker image.

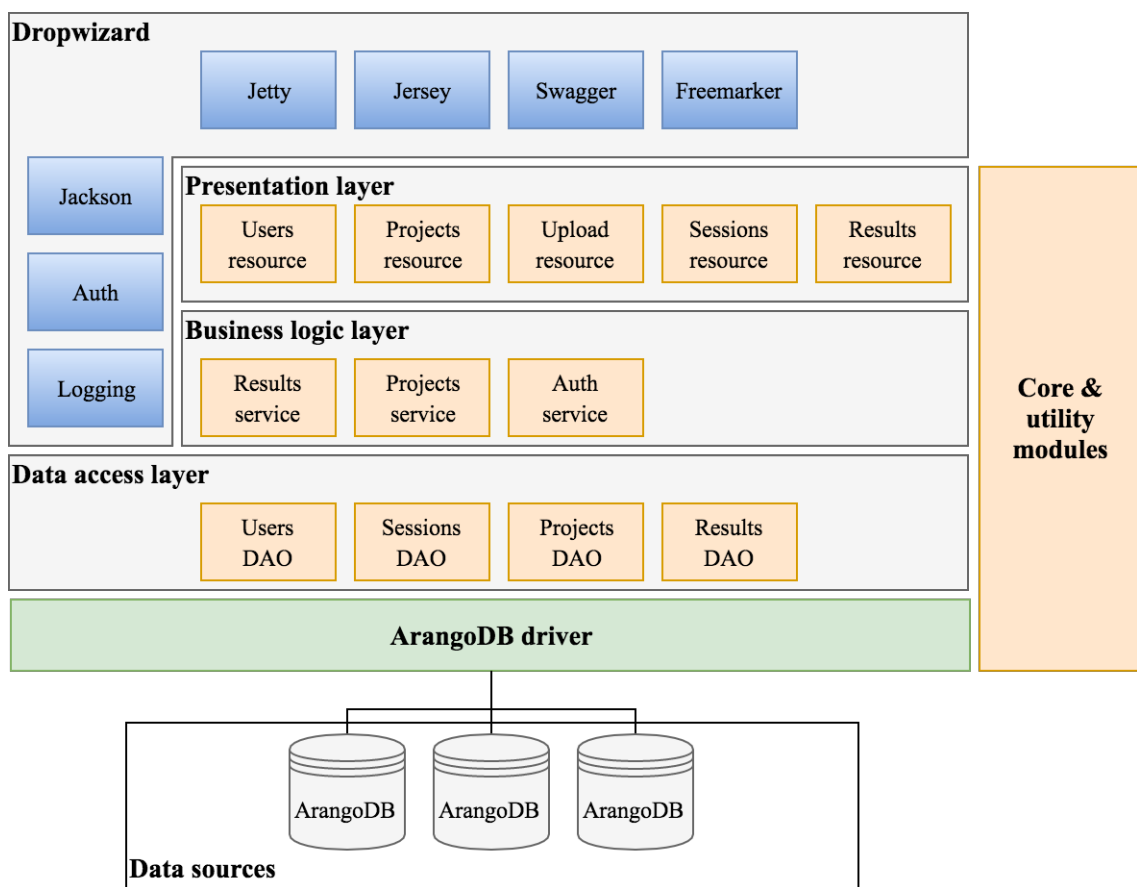


FIGURE 14. Layered architectural view

Jersey library, also bundled in Dropwizard, simplifies the development of RESTful API. Jersey itself is a reference implementation from Java API for RESTful Web Services (JAX-RS) and provides a set of Java annotations to describe a Java class as a web resource. Some of the Java class and method annotations provided by JAX-RS and Jersey to describe endpoints are listed in table 1.

TABLE 1. JAX-RS/ Jersey annotations (Jersey 2017).

| Annotation                 | Description  |
|----------------------------|--|
| @Path                      | Specifies the URI path of the resource.                            |
| @GET, @PUT, @POST, @DELETE | Specifies the request type of the resource.                        |
| @Produces                  | Specifies the MIME media type of resource produces back to client. |
| @Consumes                  | Specifies the MIME media type that resource can consume.           |
| @PathParam                 | Binds the method parameter to URI path segment.                    |
| @QueryParam                | Binds the method parameter to HTTP query parameter.                |
| @FromParam                 | Binds the method parameter to a form value.                        |

These annotations are being used throughout the resource endpoint classes in the representation layer to describe the API (figure 14). The API consumes and produces mainly JSON formatted data and for this purpose Dropwizard comes with Jackson library to bind Java objects to and from JSON.

The business logic in the service is separated in to service classes on business logic layer. There is a service for each core feature such as one for processing and querying results data and one for managing user authorization. Service for processing results data from input format, such as XML into JSON uses Freemarker library to map data from format to another.

For interfacing with ArangoDB, the application uses Java implementation of the database driver. The driver uses VelocityStream binary protocol to connect to database. Also, the ArangoDB driver supports serializing and de-serializing Java objects to and from JSON. Interfaces with the driver are separated into classes on data access layer.



## 4.2 Main system functions and features

The system has two main features that are exposed to its clients through a HTTP REST API. Firstly, it allows clients to upload test results using the API, processes them into acceptable format and inserts into database. Secondly, stored results can be later fetched and generated into reports.

### 4.2.1 Storing test results

Figure 15 illustrates the flow of uploading processing and storing test results. The client in, figure 15, is Jenkins CI server that primarily uploads results after a test stage or at the end of the pipeline. Upload requests shall also include context information about the CI build that is also stored into database and can be used to query results and generate reports.

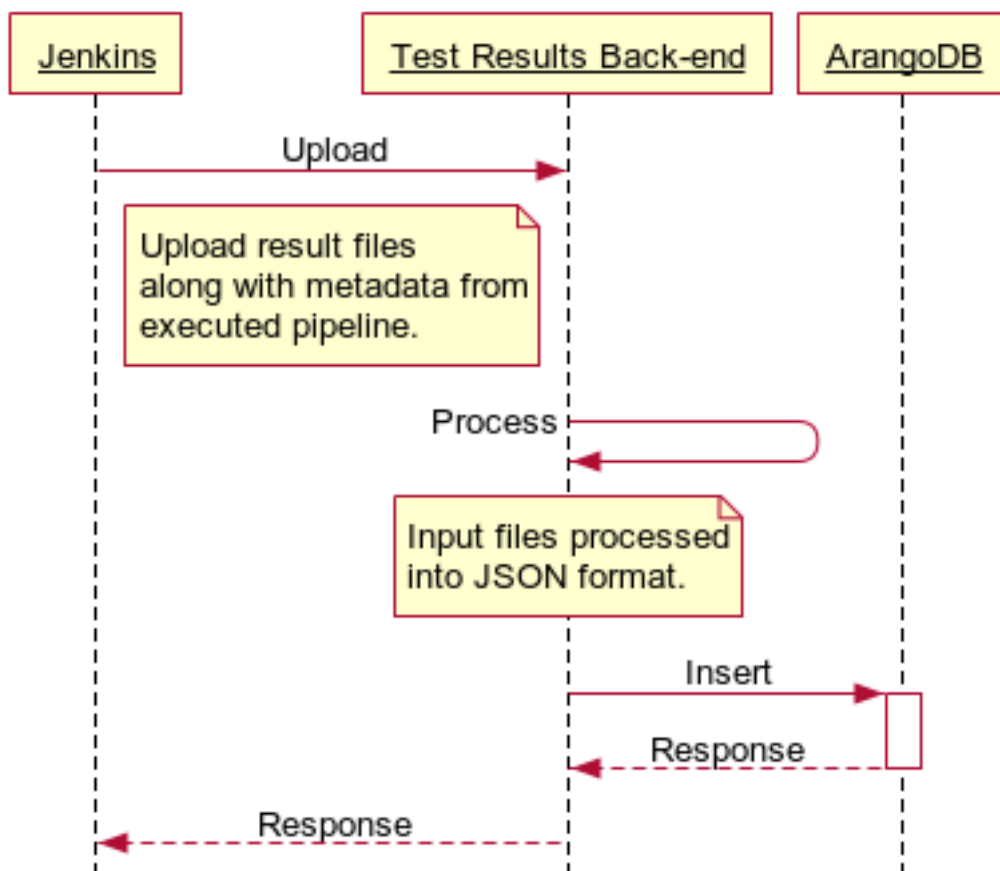


FIGURE 15. Test results upload, process and insert flow.

The API provides HTTP POST endpoint that consumes multipart/form-data content type. This media type allows embedding possibly multiple files into request along with other the data passed as form fields. Internally, service processes imported files to normalized JSON and stores documents into database and inserts new test session document containing build, test run and environment information.

#### 4.2.2 Fetching test reports

Test sessions can be searched globally or making a scoped search to include only particular project's sessions. In both cases, filtering is possible to tell API to return only a subset of results (figure 16).

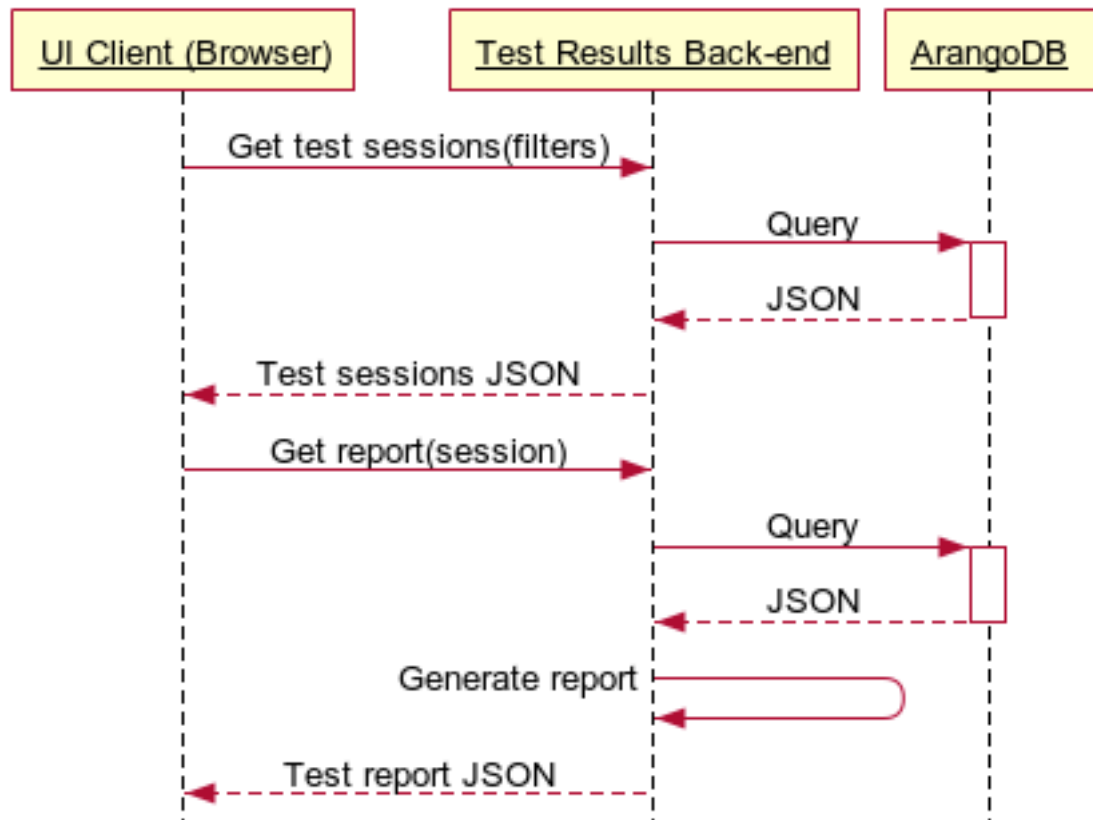


FIGURE 16. Searching for results and generating reports

The API endpoints targeted for search operations are primarily producing JSON data. A test report requested from API is generated dynamically by doing ArangoDB level joins into JSON documents. These operations combine the test session and results data into one JSON document sent to the client. The example report is shown in Appendix 1.

### 4.3 Data model

The test session and results data is stored into ArangoDB as JSON objects and grouped into document collections. Although ArangoDB does not force to use normalized JSON data, in practice the documents within same collection would end up having the same attributes belonging to the same collection.

There are two types of document collection in ArangoDB: *Document* or *vertex* collections and *vertex* and *edge* collections that are used to create relations between documents (arangodb.com 2017). Documents may one-to-many or many-to-many relations created by edges. All documents contain special attributes that are embedded into JSON object by ArangoDB. A document handle (*\_id* attribute) uniquely identifies a document in the database and consists of collection name and a document key (*\_key* attribute). Document key uniquely identifies a document in a collection. The key values are also indexed which makes looking up documents by key a fast operation (arangodb.com 2017). Documents also have a *\_rev* attribute specifying document revision and maintained by ArangoDB automatically.

The service in scope of this thesis organizes data into three document collections: *projects*, *test sessions* and *test results* (figure 17) and the relationships of data is described in the edge collections connecting documents.

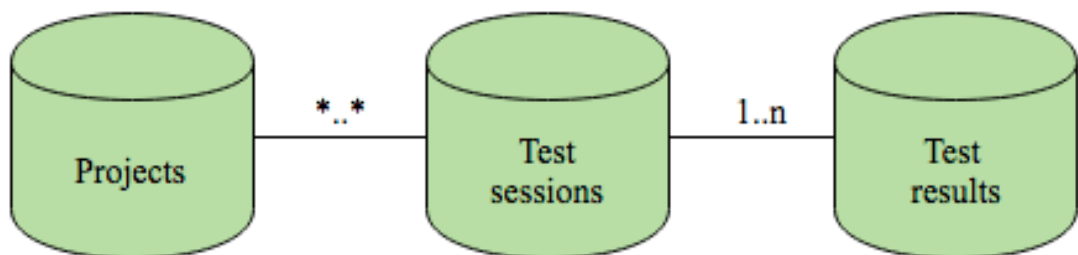


FIGURE 17. Document collections

The *projects* collection holds the JSON objects describing the projects, which test results can link to via test sessions. A *test session* is a document that contains information about the build and test execution such as build id, test environment and so on. Projects and test session can have a many-to-many relations meaning that a test session can relate to multiple projects and similarly a project may link to several test sessions. The

relationship between test sessions and test results is one-to-many. A test result document holds information about the executed test case such as name, status or duration. A test session shall always link to at least one result document. Edge collections connecting documents must have mandatory parameters *\_from* and *\_to*, which tell ArangoDB the linked document id's and the direction. ArangoDB uses edges to traverse documents when targeting queries to documents.

#### 4.4 API overview

The RESTful API's implemented in the service are listed in table 2. The three most important API's are presented in this chapter in more detailed. These are *upload*, *projects* and *sessions* API.

TABLE 2. HTTP API descriptions

| API      | Description   |
|----------|---|
| Admin    | Provides a set of management operations such as validating, if file format is supported by the service.                         |
| Upload   | Provides the endpoint for uploading and storing test result files from CI/ CD server or test reports UI.                        |
| Projects | API for altering projects that test results can be linked to. Provides also search operations to project specific test results. |
| Sessions | Provides API search operations targeted to test session's data.   |
| Results  | API for altering individual test results documents e.g. to be able to manually modify a result of a test case from UI.          |
| Users    | A simple API for managing API users and issuing authentication tokens.  |

All APIs are consuming a JSON request body, except the upload API, which accepts multipart/form-data content type.

#### 4.4.1 Upload API

The *upload* resource endpoint is used to import new test results from CI. The endpoint supports HTTP POST method to request a multipart/form-data file upload with additional form parameters. Choosing this type of method to upload files is perfect for the use case as it is generic and can be done for example using a *curl* command line tool or from any programming languages HTTP library.

The information about CI builds and code changes, are sent as part of the HTTP POST request as form parameters. From the parsed form parameters, service is creating a new test session JSON object that is stored in to ArangoDB with the test results and linked using edge documents.

#### 4.4.2 Projects API

Uploaded test results can be optionally linked to project entities that have been created to database prior to importing any results. API provides a *projects* endpoint for managing project entities and searching for projects' test sessions and results. A search operation can be targeted to all project's sessions or performing a scoped search to only particular version's results.

TABLE 3. Projects API description

| API Operation           | Description  | Method |
|-------------------------|--|--------|
| Create project          | Creates a new project entity document into database.     | POST   |
| Get project             | Fetch a project document by name.                        | GET    |
| Delete project          | Removes project and all linked documents.                | DELETE |
| List projects           | List projects by using optional filtering.               | GET    |
| Get project test report | Fetch test reports linked to a specific project version. | GET    |
| Update project          | Update existing project.                                 | PUT    |

### 4.4.3 Test sessions API

Test sessions API is used to globally search for any stored sessions, linked projects and results. This API also has an */report* endpoint that can be used to fetch a JSON including this information in single document. Internally service uses ArangoDB's graph capabilities to consolidate information from different collection and documents that have been linked to each other by edges.

TABLE 4. Sessions API description

| API Operation                | Description  | Method |
|------------------------------|--|--------|
| Get test session document    | Fetch a single test session document.                            | GET    |
| List test sessions           | List test session documents using filtering.                     | GET    |
| Fetch a test session report  | Fetch a test session report document.                            | GET    |
| Delete test session document | Delete test session and all linked results.                      | DELETE |
| Get test session summary     | Get a test session summary report incl. some metrics from tests. | GET    |

## 4.5 Security

The resources that alter data such as POST, PUT or DELETE, have been protected requiring an authentication token in the request. Technology used here to authorize user request is JSON Web Token (JWT). It is an open standard defining a compact and self-contained way to secure transmitting information as a JSON object. JWTs are digitally signed and therefore can be trusted and verified (JWT 2017).

A common use case for using JWT is authentication. In this scenario requests shall include the JWT allowing user to access resources that are permitted with the token. In authentication, when user has logged in a token is issued and returned to the client. Whenever client makes a request to a protected resource, JWT should be sent as well. It

is typically included in the *Authorization* header using the *Bearer* schema. Server's protected resources check the validity of JWT from the authorization header. If JWT is present and, valid access is granted to the protected resource. (JWT 2017)

Fortunately, there are libraries for Java already providing functionality for creating and verifying JWTs. In this project the Java JWT (JJWT) library was used due to its simplicity to use. In the Java classes, the protected resources were secured by adding a custom annotation to methods that would block requests of reaching the resources in the case of a missing or invalid JWT.

## 4.6 Deployment

The deployment of the web service is done by using Docker. Although the initial deployment was done on a virtual machine cluster hosting Docker engines in the future it would be most probably be deployed onto a cloud OS instance. Dockerizing a Dropwizard application is simple, when it is packaged as an executable Java Archive including all its dependencies. Then only this single JAR needed to be added inside the Docker image.

## 5 CONCLUSIONS

The scope of this thesis was to design and partially implement a system that provides functionality to store test results from Jenkins continuous delivery pipeline and enable building of user interfaces by providing necessary APIs to execute queries over data. Eventually, the ultimate goal was to improve the current test reports generated from automated testing in the case project, which were merely static documents by enabling more dynamic reports.

One of the key ideas in starting to build the web service, was to make it generic enough to support the existing well-known test result formats and which could be later extended to support new formats with a small implementation effort. As the most test frameworks and tools in use already supported xUnit formatted XML, this was a natural format to start with. Another option would have been to design a new generic test results file format, but this would mean to implement a results writer for each test framework in use, but this would have been too big effort.

The second key idea was to use a NoSQL ArangoDB in the back-end to store test results data. This path was chosen, because test results and reports are documents by nature and thus it felt natural to use a document database for this purpose. The fact that ArangoDB provides the graph functionality to create relationships between documents in different collections supported this idea well too.

The technology that was chosen to implement the web service ended up being Java and Dropwizard framework. A NodeJS based JavaScript project could have worked as well, because for all the Java libraries being used there was also JavaScript counterpart. Actually, the main thing impacting to the selection was that Java was more familiar to the writer.

Although the web service has now been implemented to cover the most important use cases to enable importing of test results from Jenkins CD pipeline and provide search capabilities for results, the work continues towards the complete reporting solution. The next phases will be integrating the service as part of the other continuous integration infra as well as starting the implementation of the reporting user interface.



The user interface design and implementation work will start from defining first views to show fast-feedback reports with traditional test execution status information along with other context data that have been stored into test session documents. Later, more views will be added to provide search capabilities. Now that the test result data is stored into database, it gives nearly endless possibilities to create different UIs to data. Some ideas discussed for future improvements can be to provide a quality dashboards and metrics UI where users can further drill down to detailed results.

The back-end web service will also be further developed to provide new API endpoints. For example, there will be a need to provide an ability to alter test execution statuses later or include comments to results or add links to external issue management systems. Also, there might be a need to provide cumulated reports or comparison that make sense to generate in the back-end. All of these will of course reflect to front-end side as well.

This thesis showed that the example solution provided here can be a flexible way of creating a reporting system for continuous integration and delivery pipeline that could be take into use for any project with some work. For example, if the test frameworks used in the project output some other result format than xUnit, a new processor class would be required to process the file into supported JSON format. NoSQL databases have also been improved a lot in the past years a suite well for this kind of use cases as they provide more similar capabilities as the relational databases.

## REFERENCES

arangodb.com. 2017 ArangoDB Documentation. Read 17.04.2017.  
<https://www.arangodb.com/>

Crispin L., Gregory J. 2014. More Agile Testing: Learning Journeys for the Whole Team. Boston: Addison-Wesley Professional. Read 26.2.02.2016.  
<https://www.safaribooksonline.com>

Dustin E., Garret T., Gauf B. 2009. Implementing Automated Software Testing; How to Save Time and Lower Costs While Raising Quality. Boston: Addison-Wesley Professional.

Dallas A. 2014. RESTful Web Services with Dropwizard. Packt Publishing. Read 18.04.2017. <https://www.safaribooksonline.com>

Docker. 2017a. What is a Container. Read 09.09.2017. <https://www.docker.com/what-container>

Docker. 2017b. Get started. Read 09.09.2017. <https://docs.docker.com/get-started/>

Dropwizard. Getting started. 2017. Read 17.04.2017. <http://www.dropwizard.io/>

Fowler M. 2002. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley Professional. Read 11.09.2016. <https://www.safaribooksonline.com>

Fowler M. 1.5.2006. Continuous Integration. Read 26.03.2017.  
<https://www.martinfowler.com/articles/continuousIntegration.html>

Fowler M. 30.5.2013. Continuous Delivery. Read 26.03.2017.  
<https://martinfowler.com/bliki/ContinuousDelivery.html>

Fowler M, Lewis J. 2014. Microservices. Read 26.08.2017.  
<https://martinfowler.com/articles/microservices.html>

Gregory J., Crispin L. 2008. Agile Testing: A Practical Guide For Testers and Agile Teams. Boston: Addison-Wesley Professional. Read 26.2.02.2016.  
<https://www.safaribooksonline.com>

Hambling B., Morgan P., Samaroo A., Thompson G., Williams P. 2015. Software Testing – An ISTQB-BCS Certified Tester Foundation 3<sup>rd</sup> edition. United Kingdom. BCS Learning & Development. Read 03.04.2017. <https://www.safaribooksonline.com>

HIPAA. 2017. Read 14.08.2017.  
<http://www.dhcs.ca.gov/formsandpubs/laws/hipaa/Pages/1.00WhatIsHIPAA.aspx>

Humble, J., Farley D. 2011. Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation. Boston: Addison-Wesley Professional.

Humble, J. 13.8.2010. Continuous Delivery vs Continuous Deployment.  
Read 30.05.2017. <https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/>

Jenkins. Meet Jenkins. 2016. Read 16.04.2017.  
<https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins>

Jenkins Pipeline. Read 17.04.2017. <https://jenkins.io/doc/book/pipeline/>

Jersey. Jersey User Guide. 2017. Read 12.09.2017.  
<https://jersey.github.io/documentation/latest/index.html>

JWT. Introduction to JSON Web Tokens. 2017. Read 10.10.2017.  
<https://jwt.io/introduction/>

Kaner C., Bach J., Pettichord B. 2002. Lessons Learned in Software Testing; A Context-Driven Approach. New York. John Wiley & Sons, Inc.

liaison.com. 2017. Liaison ALLOY Platform. Read 05.04.2017.  
<https://www.liaison.com>

opensource.com. 2017. What is Docker? 09.09.2017.  
<https://opensource.com/resources/what-docker>

PCI Security. 2017. Standards overview. Read 14.08.2017.  
<https://www.pcisecuritystandards.org/>

## APPENDICES

### Appendix 1. Example test report JSON document

1(2)

```
{
  "testSession": {
    "_key": "string",
    "title": "string",
    "environment": "string",
    "testType": "string",
    "stage": "string",
    "buildId": "string",
    "buildUrl": "string",
    "repositoryUrl": "string",
    "repositoryType": "string",
    "commitId": "string",
    "scmBranch": "string",
    "labels": [
      "string"
    ],
    "created": "string",
    "source": "string"
  },
  "projects": [
    {
      "project": {
        "name": "string",
        "description": "string",
        "labels": [
          "string"
        ],
        "created": "string",
        "updated": "string"
      },
      "version": "string",
      "roleType": "string"
    }
  ],
  "testSuites": [
    {
      "_key": "string",
      "created": "string",
      "updated": "string",
      "name": "string",
```

**Appendix 1. Example test report JSON document**

2 (2)

```

    "timestamp": "string",
    "duration": 0,
    "testsCount": 0,
    "failureCount": 0,
    "passedCount": 0,
    "skippedCount": 0,
    "errorCount": 0,
    "testProperties": [
      {
        "name": "string",
        "value": "string"
      }
    ],
    "testResults": [
      {
        "name": "string",
        "details": "string",
        "duration": 0,
        "status": "string",
        "failureDetails": "string",
        "failureStackTrace": "string",
        "errorDetails": "string",
        "errorStackTrace": "string",
        "stderr": "string",
        "comment": "string",
        "issues": [
          {
            "issueId": "string",
            "issueType": "string",
            "issueUrl": "string"
          }
        ]
      }
    ]
  }
}

```